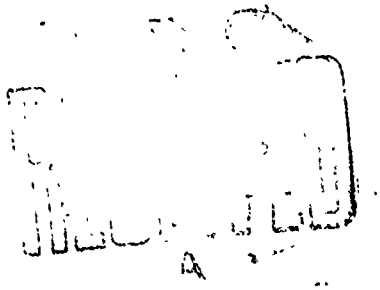
# FINDING THE TRICONNECTED COMPONENTS OF A GRAPH

J.E. Hopcroft and R.E. Tarjan

TR 72 - 140



August 1972



Computer Science Department
Cornell University
Ithaca, New York 14850

Security Classification

DOCUMENT CONTROL DATA - R & D

*(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)*

| 1. ORIGINATING ACTIVITY (Corporate author) | 2a. REPORT SECURITY CLASSIFICATION |
|---|---|
| Cornell University | Unclassified |
| | 2b. GROUP |

3. REPORT TITLE

FINDING THE TRICONNECTED COMPONENTS OF A GRAPH

4. DESCRIPTIVE NOTES (Type of report and, inclusive dates)

Technical Report

5. AUTHOR(S) (First name, middle initial, last name)

John E. Hopcroft and Robert E. Tarjan

| 6. REPORT DATE | 7a. TOTAL NO. OF PAGES | 7b. NO. OF REFS |
|---|---|---|
| August 1972 | 61 | |

| 8a. CONTRACT OR GRANT NO. | 9a. ORIGINATOR'S REPORT NUMBER(S) |
|---|---|
| Hertz Foundation | 72-140 |
| b. PROJECT NO. N00014-67-A-0077-0021 | |
| c. | 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) |
| d. | None |

10. DISTRIBUTION STATEMENT

Approved for public release, distribution unlimited

| 11. SUPPLEMENTARY NOTES | 12. SPONSORING MILITARY ACTIVITY |
|---|---|
| | Office of Naval Research, Rochester |

13. ABSTRACT

An algorithm for decomposing a graph into triconnected components is presented. The algorithm requires $O(|V|+|E|)$ time and space when implemented on a random access computer, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. The algorithm is both theoretically optimal (to within a constant factor) and efficient in practice.

1

DD FORM 1 NOV 68 1473 (PAGE 1)

S/N 0101-807-6811

Security Classification

| 14. KEY WORDS | LINK A | | LINK B | | LINK C | |
|---|---|---|---|---|---|---|
| | ROLE | WT | ROLE | WT | ROLE | WT |
| articulation point | | | | | | |
| backtracking | | | | | | |
| connectivity | | | | | | |
| depth-first search | | | | | | |
| graph | | | | | | |
| separability | | | | | | |
| separation | | | | | | |
| triconnectivity | | | | | | |

# FINDING THE TRICONNECTED COMPONENTS OF A GRAPH

J.E. Hopcroft and R.E. Tarjan[†]

Abstract:

An algorithm for decomposing a graph into triconnected components is presented. The algorithm requires $O(|V|+|E|)$ time and space when implemented on a random access computer, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. The algorithm is both theoretically optimal (to within a constant factor) and efficient in practice.

Keywords: articulation point, backtracking, connectivity, depth-first search, graph, separability, separation, triconnectivity

---

[†]This research was supported in part by the Hertz Foundation and the office of Naval Research under grant N00014-67-A-0077-0021.

## Introduction

The connectivity properties of graphs form an important part of graph theory. Efficient algorithms for determining the connectivity structure of graphs are both theoretically interesting and useful in a variety of applications. One technique which may be used to solve connectivity problems is that of backtracking, or depth-first search. In [10] depth-first search is applied to give efficent algorithms for determining the biconnected components of an undirected graph and for determining the strongly connected components of a directed graph. This paper extends the application of depth-first search to the problem of finding the triconnected components of a graph.

An algorithm for determining the triconnected components of a graph is needed by procedures for determining whether a graph is planar [2] and for determining whether two planar graphs are isomorphic [8]. Standard methods for determining the triconnected components of a graph require $O(|V|^3)$ steps or more, if the graph has $|V|$ vertices. The algorithm described here requires substantially less time, and may be shown to be optimal to within a constant factor assuming a suitable model of computation.

This paper is divided into four sections. The first section presents the necessary definitions and lemmas from graph theory. The theory of the triconnected components of a graph was developed by Tutte [11]. A modified exposition more

suitable for computer applications is given here. The theory is also a special case of the more general theory of decomposing "clutters" into chunks due to Edmonds and Cunningham [ 5]. The second section describes depth-first search and the data structures needed to implement it efficiently on a computer. The third section describes preliminary calculations and a simple test to find the separation pairs of a graph. The last section describes the heart of the triconnected components algorithm, including proofs of its correctness and time and space bounds.

In deriving time bounds on algorithms we assume a random access model. A formal definition of such a model may be found in [ 4 ]. We use the following notation for specifying bounds of algorithms: if $\vec{n}$ is a vector, f is a real-valued function, and there exist constants $k_1$, $k_2$ such that $|t(\vec{n})| \leq k_1|f(\vec{n})| + k_2$, then we write "t(n) is $O(f(\vec{n}))$".

## Graphs, Trees and Connectivity

A graph $G = (V,E)$ consists of a set of vertices $V$ and a set of edges $E$. If the edges are ordered pairs $(v,w)$ of vertices, the graph is directed; $v$ is called the tail and $w$ the head of the edge. If the edges are unordered pairs of vertices, also denoted by $(v,w)$, the graph is undirected. If $E$ is a multiset; that is, any edge may occur several times, then $G$ is a multigraph. If $(v,w)$ is an edge of a multigraph $G$, vertices $v$ and $w$ are adjacent. Edge $(v,w)$ is incident to vertices $v$ and $w$; $v$ and $w$ are incident to $(v,w)$. If $E'$ is a set of edges in $G$, $V(E')$ is the set of vertices incident to one or more of the edges in $E'$. If $S$ is a set of vertices in $G$, $E(S)$ is the set of edges incident to at least one vertex in $S$.

If $G$ is a multigraph, a path $p : v \overset{*}{\Rightarrow} w$ in $G$ is a sequence of vertices and edges leading from $v$ to $w$. A path is simple if all its vertices are distinct. A path $p : v \overset{*}{\Rightarrow} v$ is a cycle if all its edges are distinct and the only vertex to occur twice in $p$ is $v$, which occurs exactly twice. Two cycles which are cyclic permutations of each other are considered to be the same cycle. The undirected version of a directed multigraph is the multigraph formed by converting each edge of the directed multigraph into an undirected edge. A multigraph is connected if every pair of vertices $v$ and $w$ in $G$ is connected by a path.

If $G = (V,E)$ and $G' = (V',E')$ are two multigraphs such that $V' \subseteq V$ and $E' \subseteq E$, then $G'$ is a subgraph of $G$. A multigraph having exactly two vertices $v,w$ and one or more edges $(v,w)$ is called a bond.

A (directed, rooted) tree $T$ is a directed graph whose undirected version is connected, having one vertex which is the head of no edges (called the root), and such that all vertices except the root are the head of exactly one edge. The relation " $(v,w)$ is an edge of $T$ " is denoted by $v \to w$. The relation "there is a path from $v$ to $w$ in $T$ " is denoted by $v \overset{*}{\to} w$. If $v \to w$, $v$ is the father of $w$ and $w$ is a son of $v$. If $v \overset{*}{\to} w$, $v$ is an ancestor of $w$ and $w$ is a descendant of $v$. The set of descendants of a vertex $v$ is denoted by $D(v)$. Every vertex is an ancestor and a descendant of itself. If $G$ is a directed multigraph, a tree $T$ is a spanning tree of $G$ if $T$ is a subgraph of $G$ and $T$ contains all the vertices of $G$.

Let $P$ be a directed multigraph, consisting of two disjoint sets of edges, denoted by $v \to w$ and $v \dashrightarrow w$. Suppose $P$ satisfies the following properties:

(i) The subgraph $T$ containing the edges $v \to w$ is a spanning tree of $P$.

(ii) If $v \dashrightarrow w$, then $w \overset{*}{\to} v$. That is, each edge not in the spanning tree $T$ of $P$ connects a vertex with one of its ancestors in $T$.

Then P is called a palm tree. The edges $v \dashrightarrow w$ are called the fronds of P.

A connected multigraph G is biconnected if for each triple of distinct vertices v,w and a in V there is a path $p : v \overset{*}{\Rightarrow} w$ such that a is not on the path p. If there is a distinct triple v,w,a such that a is on every path $p : v \overset{*}{\Rightarrow} w$, then a is called a separation point (or an articulation point) of G. We may partition the edges of G so that two edges are in the same block of the partition if and only if they belong to a common cycle. Let $G_i=(V_i,E_i)$ where $E_i$ is the set of edges in the $i^{th}$ block of the partition and $V_i=V(E_i)$. Then:

(i) Each $G_i$ is biconnected.

(ii) No $G_i$ is a proper subgraph of a biconnected subgraph of G.

(iii) Each vertex of G which is not an articulation point of G occurs exactly once among the $V_i$ and each articulation point occurs at least twice.

(iv) For each $i,j,i\neq j,V_i \cap V_j$ contains at most one vertex; furthermore, this vertex (if any) is an articulation point.

The subgraphs $G_i$ of G are called the biconnected components of G. The biconnected components of G are unique.

Let {a,b} be a pair of vertices in a biconnected multigraph G. Suppose the edges of G are divided into equivalence classes $E_1,E_2,\ldots,E_n$ such that two edges which lie in a

common path not containing any vertex of $\{a,b\}$ are in the same class. The classes $E_i$ are called the separation classes of $G$ with respect to $\{a,b\}$. If there are at least two separation classes, then $\{a,b\}$ is a separation pair of $G$ unless (1) there are exactly two separation classes and one class consists of a single edge or (2) there are exactly three classes each consisting of a single edge.

If $G$ is a multigraph such that no pair $\{a,b\}$ is a separation pair of $G$, then $G$ is triconnected. Let $\{a,b\}$ be a separation pair of a biconnected multigraph $G$. Let the separation classes of $G$ with respect to $\{a,b\}$ be $E_1, E_2, \ldots, E_n$. Let $E' = \bigcup_{i=1}^{k} E_i$ and $E'' = \bigcup_{i=k+1}^{n} E_i$ be such that $|E'| \geq 2$, $|E''| \geq 2$. Let $G_1 = (V(E'), E' \cup \{(a,b)\})$, $G_2 = (V(E''), E'' \cup \{(a,b)\})$. The graphs $G_1$ and $G_2$ are called the split graphs of $G$ with respect to $\{a,b\}$. Replacing a multigraph $G$ by two split graphs is called splitting $G$. There may be many possible ways to split a graph, even with respect to a fixed separation pair $\{a,b\}$. A splitting operation is denoted by $s(a,b,i)$; $i$ is a label distinguishing this split operation from other splits. The new edges $(a,b)$ added to $G_1$ and $G_2$ are called virtual edges; they are labelled to identify them with the split. A virtual edge $(a,b)$ associated with split $s(a,b,i)$ will be denoted by $(a,b,i)$. If $G$ is biconnected, then any split graph of $G$ is also biconnected.

Suppose a multigraph G is split, the split graphs are split, and so on, until no more splits are possible (each graph remaining is triconnected). The graphs constructed in this way are called the split components of G. The split components of a multigraph are not necessarily unique.

Lemma 1: Let $G = (V,E)$ be a multigraph with $|E| \geq 3$. Let $G_1, G_2, \ldots, G_m$ be the split components of G. Then the total number of edges in $G_1, G_2, \ldots, G_m$ is bounded by $3|E|-6$.

Proof: By induction on the number of edges of G. If G has 3 edges the lemma is immediate, because G cannot be split. Suppose the lemma is true for graphs with n-1 edges and suppose G has n edges. If G cannot be split the lemma is true for G. Suppose on the other hand that G can be split into $G'$ and $G''$, where $G'$ has k+1 edges and $G''$ has n-k+1 edges for some $2 \leq k \leq n-2$. By induction, the total number of edges in $G_1, G_2, \ldots, G_m$ must be bounded by $3(k+1) - 6 + 3(n-k+1) - 6 = 3n - 6$. Thus by induction the lemma is true.

In order to get unique triconnected components we must partially reassemble the split components. Suppose $G_1 = (V_1,E_1)$ and $G_2 = (V_2,E_2)$ are two split components both containing a virtual edge $(a,b,i)$. Let

$$G = (V_1 \cup V_2,\ (E_1 - \{(a,b,i)\}) \cup (E_2 - \{(a,b,i)\})) .$$

Then G is called a merge graph of $G_1$ and $G_2$; the merge operation will be denoted by m(a,b,i). Merging is the inverse of splitting; if we perform a sufficient number of merges on the split components of a multigraph we recreate the original multigraph.

The split components of a multigraph are of three types: triple bonds, of the form $(\{a,b\},\{(a,b),(a,b),(a,b)\})$; triangles, of the form $(\{a,b,c\},\{(a,b),(a,c),(b,c)\})$; and triconnected graphs. Let G be a multigraph whose split components are a set of triple bonds $\mathcal{B}_3$, a set of triangles $\mathcal{T}$, and a set of triconnected graphs $\mathcal{G}$. Suppose the triple bonds $\mathcal{B}_3$ are merged as much as possible to give a set of bonds $\mathcal{B}$, and that the triangles $\mathcal{T}$ are merged as much as possible to give a set of polygons $\mathcal{P}$. Then the set of graphs $\mathcal{B} \cup \mathcal{P} \cup \mathcal{G}$ is the set of triconnected components of G. If G is an arbitrary multigraph, the triconnected components of the biconnected components of G are called the triconnected components of G. This set of components is unique, as we shall see below.

Let G be a multigraph and let $\mathcal{G}$ be a set of graphs obtained from G by a sequence of splits and merges. Consider the auxiliary graph $S(\mathcal{G})$ whose vertices are the graphs in $\mathcal{G}$. Graphs $G_1$ and $G_2$ are joined by an edge if and only if they share a common virtual edge.

Lemma 2: If $\mathcal{G}$ is a set of graphs obtained from a connected multigraph G by a sequence of splits and merges, then the auxiliary graph $S(\mathcal{G})$ is a tree.

Proof: The proof is by induction on the length of the sequence of splits and merges. For a sequence of length zero, $S(\mathcal{G})$ is a single vertex and hence a tree. Assume $S(\mathcal{G})$ is a tree for all sequences of length less than i, $i \geq 1$, and let x be a sequence of length i. Let $S'(\mathcal{G})$ be the auxiliary graph after the first i-1 splits and merges in the sequence x. By the induction hypothesis $S'(\mathcal{G})$ is a tree. Assume x ends with a split. Then $S(\mathcal{G})$ is obtained from $S'(\mathcal{G})$ by replacing a vertex v by two vertices $v'$ and $v''$ connected by an edge $(v',v'')$. Each edge $(u,v)$ in $S'(\mathcal{G})$ is replaced by an edge $(u,v')$ or by an edge $(u,v'')$ depending on whether the virtual edge common to u and v is in the split component $v'$ or $v''$. $S(\mathcal{G})$ minus the edge $(v',v'')$ consists of two trees, one containing $v'$ and one containing $v''$. Hence $S(\mathcal{G})$ is a tree. Assume x ends with a merge. Then two vertices $v'$ and $v''$ in $S'(\mathcal{G})$ which are connected by an edge are replaced by a single vertex v, the edge $(v',v'')$ is deleted and each edge $(u,v')$ or $(u,v'')$ is replaced by an edge $(u,v)$. Since $S'(\mathcal{G})$ is a tree, $S'(\mathcal{G})$ with edge $(v',v'')$ deleted consists of two trees, one containing $v'$ and one containing $v''$. Thus $S(\mathcal{G})$ is a tree.

Lemma 3: Let G be a biconnected multigraph. Suppose a sequence of intermixed splits and merges is carried out on G. Then there is a sequence of splits which produces the same set of graphs.

Proof: Consider any intermixed sequence of splits and merges Let m(a,b,i) be the first merge. A split s(a,b,i) must have been previously performed. Then deleting s(a,b,i) and m(a,b,i) does not affect the set of graphs produced, because all splits performed before m(a,b,i) may still be performed. This gives an equivalent sequence with one less merge. The lemma follows by induction.

Lemma 4: Let a,b,c,d be distinct vertices in a biconnected multigraph G. Suppose {a,b} and {c,d} are separation pairs, and that some split s(a,b,i) of G puts c into one split graph and d into the other. Then every split s(c,d,j) of G puts a into one split graph and b into the other.

Proof: Suppose that contrary to the lemma, some split s(c,d,j) with split graphs $G_1$ and $G_2$ has both a and b in the same split graph, say $G_1$. There must exist a path between c and d in $G_2$ which does not contain the virtual edge (c,d,j). This path contains neither a nor b since both are in $G_1$. Thus there is a path in G containing neither a nor b, but connecting c and d. This is impossible since split s(a,b,i) puts

c into one split graph and d into the other. Thus the lemma must be true.

Lemma 5: Let a,b,c,d be distinct vertices in a biconnected multigraph G. Suppose {a,b} and {c,d} are separation pairs and that some split s(a,b,i) puts c in one split graph $G_1$ and d in the other split graph $G_2$. Then either {a,c} is a separation pair in $G_1$, or there are exactly two edges incident to a in $G_1$; namely, (a,c) and the virtual edge (a,b,i).

Proof: As a consequence of Lemma 4, each path from a to b in G contains either c or d. Hence, each path from a to b in $G_1$ either contains c or is the virtual edge (a,b,i). (In particular, there is only one edge between a and b.) Suppose $G_1$ contains a vertex $v \neq b,c$ with v adjacent to a. Since $G_1$ is biconnected there is a simple path from v to b not containing a. Any such path must contain the vertex c and hence {a,c} must be a separation pair. If b and c are the only vertices adjacent a, then either the edge (a,c) is a multiple edge in which case {a,c} is a separation pair, or there are exactly two edges incident to a, namely (a,c) and the virtual edge (a,b,i).

Theorem 6: The triconnected components of a multigraph G are unique.

Proof: Lemma 3 shows that given any sequence of splits and merges there is an equivalent sequence consisting solely of splits which yield the same set of triconnected components. Thus we need only show that any two sequences of splits which yield sets of triconnected components, yield the same set.

The proof is by induction on the number of edges in the multigraph $G$. If $G$ has fewer than four edges, the theorem is certainly true. Assume the theorem is true for graphs with less than $k$ edges. Let $G$ have $k$ edges. If $G$ has no separation pairs, the theorem is true for $G$. Thus suppose $G$ has at least one separation pair.

If the first split is the same for each sequence, then the split graphs after the first split are the same for each sequence. Since each split graph has fewer edges than the original graph, the theorem is true on the split graphs by the induction hypothesis. Thus the theorem is true for the entire graph.

Suppose the first split in one sequence is $s(a,b,i)$, giving split graphs $G_1$ and $G_2$, and the first split in the other sequence is $s(c,d,j)$, giving split graphs $G_3$ and $G_4$. We perform a case analysis.

Case 1: $\{a,b\} = \{c,d\}$. Since neither sequence contains a merge the triconnected components resulting from the first sequence must be the union of the triconnected components of $G_1$ and $G_2$ and the triconnected components

resulting from the second sequence must be the union of the triconnected components of $G_3$ and $G_4$.

Suppose {a,b} is a separation pair in both $G_1$ and $G_2$. By the induction hypothesis we may split and merge $G_1$ and $G_2$ in an arbitrary order to get their triconnected components. If we split $G_1$ first on {a,b}, we produce a split component which contains a double edge (a,b). Splitting again on {a,b} produces a triple bond (a,b). Similarly, splitting $G_2$ twice on {a,b} produces a triple bond (a,b). These bonds must be merged to give the triconnected components of G. But this is a contradiction, since the original sequence of splits had no merge m(a,b,i).

Thus the set {a,b} cannot be a separation pair for both $G_1$ and $G_2$. Similarly {a,b} cannot be a separation pair for both $G_3$ and $G_4$. Without loss of generality assume {a,b} is not a separation pair for $G_1$ or $G_3$. Let $E_i$, $1 \leq i \leq 4$, be the edge set of $G_i$. There exist disjoint sets of edges $E_1'$, $E_3'$ and E which are unions of the separation classes of G with respect to {a,b} such that

$$E_1 = E_1' \cup \{(a,b,i)\} \qquad E_2 = E_3' \cup E \cup \{(a,b,i)\}$$
$$E_3 = E_3' \cup \{(a,b,j)\} \qquad E_4 = E_1' \cup E \cup \{(a,b,j)\}$$

By the induction hypothesis we may apply any sequence of splits to $G_2$ and $G_4$ to give their triconnected components. Split $G_2$ using a split s(a,b,k) putting $E_3'$ into one split graph and $E \cup \{(a,b,i)\}$ into the other. Split $G_4$

using a split $s(a,b,\ell)$ putting $E_1'$ into one split graph and $E \cup \{(a,b,j)\}$ into the other. The two resulting sets of graphs are isomorphic and all components have fewer than $k$ edges. It then follows from the induction hypothesis that in this case the triconnected components are unique.

Case 2: $\{a,b\} \neq \{c,d\}$. Without loss of generality assume $G_1$ contains $c$ and $d$ or that neither $G_1$ nor $G_2$ contains both $c$ and $d$. In the latter case assume that $c$ is in $G_1$ and that $d$ is in $G_2$. We consider each assumption separately.

a) Assume $G_1$ contains $c$ and $d$. Without loss of generality assume $G_3$ contains $a$ and $b$, since if neither $G_3$ nor $G_4$ contains $a$ and $b$, $a,b,c$ and $d$ must all be distinct and Lemma 4 implies that $G_1$ cannot contain both $c$ and $d$. Pair $\{c,d\}$ must be a separation pair in $G_1$ and $\{a,b\}$ must be a separation pair in $G_3$. Apply split $s(c,d,k)$ to $G_1$ and $s(a,b,\ell)$ to $G_3$. The two resulting sets of graphs are isomorphic and all components have fewer than $k$ edges. It then follows from the induction hypothesis that in this case the triconnected components are unique.

b) Assume neither $G_1$ nor $G_2$ contains both $c$ and $d$ and that $c$ is in $G_1$ and $d$ is in $G_2$. Clearly

a,b,c and d must be distinct. Since the first sequence does not contain a merge the triconnected components of G must be the union of the triconnected components of $G_1$ and $G_2$. Since $G_1$ and $G_2$ each contain fewer than k edges we can obtain their triconnected components by any sequence of splits and merges. By Lemma 5 either (a,c) is a separation pair of $G_1$ or there are exactly two edges incident at a, namely (a,c) and (a,b,i).

A similar statement holds for the pair (b,c). Thus if $G_1$ is not already a triangle with a virtual edge (a,b), a triangle can be obtained by splits with respect to (a,c) or (b,c) or both. Thus, $G_1$ must have a triconnected component which is a polygon containing the virtual edge (a,b). A similar argument implies that $G_2$ also has a triconnected component which is a polygon containing the virtual edge (a,b). This contradicts the claim that the triconnected components of G are the union of the triconnected components of $G_1$ and $G_2$, since these polygons may be merged. We conclude that this case cannot arise.

All cases are covered by the above arguments. Thus by induction the theorem is true.

The triconnected components of multigraph G = (V,E) are unique. By Lemma 3, it is possible to construct the triconnected

components of a graph G using only split operations and no merge operations. Tutte [11] has formulated a particular type of split which is suitable for this purpose. Let {a,b} be a separation pair of G. Let C be one separation class of a biconnected multigraph G with respect to {a,b}, and let $\overline{C} = E - C$. Suppose $|C| \geq 2$, $|\overline{C}| \geq 2$, and either $(V(C),C)$ is biconnected or $(V(\overline{C}),\overline{C})$ is biconnected. Then we may apply a Tutte split $s(a,b,i)$ to G, producing split graphs $G_1 = (V(C), C \cup \{(a,b,i)\})$, $G_2 = (V(\overline{C}), \overline{C} \cup \{(a,b,i)\})$. The Tutte components of a biconnected multigraph G are the graphs found by applying a Tutte split to G, applying Tutte splits to the split graphs, and repeating the process until no Tutte splits are possible. The Tutte components of an arbitrary multigraph G are the Tutte components of the biconnected components of G.

Lemma 7: Let G be a biconnected multigraph and let $\alpha = (a,b)$ be a fixed edge of G. Assume $G_1$ contains the edge $\alpha$ and arises from G by a sequence of splits. If c is an articulation point of $G_1$ minus the edge $\alpha$, then c is an articulation point of G minus the edge $\alpha$.

Proof: The proof is by induction on the length of the sequence of splits. Consider a sequence of length one. Let G give rise to $G'$ and $G''$ by the split $s(d,e,i)$ and let c be an articulation point of $G'$ minus the edge $\alpha$. There exist f and g in $G'$ such that every path from

f to g in $G'$, not containing $\alpha$, contains the vertex c. If a path from f to g in G did not contain c, the path must consist of three segments, two in $G'$ and one a path from d to e in $G''$. The segment from d to e can be replaced by the virtual edge (d,e,i) to give a path from f to g in $G'$ not containing c. This is a contradiction and we conclude that c is an articulation point of G. The induction step follows immediately by dividing each sequence of length k into two sequences, one of length one and the other of length k-1.

Lemma 8: The Tutte components of a multigraph G are identical to the triconnected components of G and are thus unique.

Proof: By Tutte [ 11 ], a multigraph has no Tutte split if and only if it is either a triconnected graph, a bond, or a polygon. Thus the triconnected components of a graph are the Tutte components, with possibly a few merges carried out among bonds and among polygons.

Suppose a Tutte split s(a,b,i) produces split graphs $G_1$ and $G_2$. Without loss of generality we can assume that $G_1$ minus the virtual edge (a,b,i) is biconnected. Assume $G_1$ gives rise to a polygon $P_1$ containing the virtual edge (a,b,i). By Lemma 7 every articulation point of $P_1$ minus the edge (a,b,i) is an articulation point of $G_1$ minus the the edge (a,b,i). Since $P_1$ minus the edge (a,b,i) is not biconnected, neither is $G_1$ minus the edge (a,b,i). Simi-

larly, if $G_2$ also gives rise to a polygon containing (a,b,i) then $G_2$ minus (a,b,i) is not biconnected. But this is impossible by the definition of a Tutte split. Thus no two polygons which are Tutte components can share a virtual edge.

Suppose a Tutte split s(a,b,i) produces split graphs $G_1$ and $G_2$. By the definition of a Tutte split one of $G_1$ or $G_2$, say $G_1$, must correspond to a single separation class with respect to {a,b} containing at least two edges. Thus $G_1$ cannot be a bond, nor can {a,b} be a separation pair of $G_1$. From this it follows that $G_1$ cannot give rise to a bond containing (a,b,i). Thus no two bonds which are Tutte components are possible, and the Tutte components of a graph are the same as the triconnected components of a graph.

Figure 1 illustrates a biconnected graph G with several separation pairs. The triconnected components of G are illustrated in Figure 2.

Depth-First Search

Backtracking, or depth-first search, is a technique which is extremely useful in determining certain properties of graphs. Suppose G is a multigraph which we wish to explore. Initially all the vertices of G are unexplored. Start from some vertex of G and choose an edge to follow. Traversing the edge leads to a new vertex. Continue in this way; at each step, select an unexplored edge leading from a vertex already reached and traverse this edge. The edge leads to some vertex, either new or already reached. Whenever there are no edges leading from old vertices, choose some unreached vertex, if any exists, and begin a new exploration from this vertex. Eventually all the edges of G will be traversed. Such a process is called a search of G.

There are many ways of exploring a graph, depending upon the way in which edges to search are selected. Consider the following choice rule: when selecting an edge to traverse, always choose an edge emanating from the vertex most recently reached which still has unexplored edges. A search which uses this rule is called a depth-first search. The set of old vertices with possibly unexplored edges may be stored on a stack. Thus a depth-first search is very easy to program either iteratively or recursively, given a suitable computer representation of a graph.

For each vertex v of G we may construct a list $A_v$ containing all vertices u such that (v,u) is an edge of G. Such a list is called an adjacency list for vertex v.

A set A of such lists, one for each vertex in G, is called an adjacency structure for G. If G is undirected, each edge (v,u) is represented twice in an adjacency structure; once for v and once for u. If G is directed, each edge (v,u) is represented once: vertex u appears in the adjacency list of vertex v. A single graph may have many adjacency structures; in fact, each ordering of the edges at the vertices of G gives a unique adjacency structure, and each adjacency structure corresponds to a unique ordering of the edges at each vertex. If G is connected, each adjacency structure and starting vertex for G determines a unique depth-first search of G, given by using the following choice rule in the search: if an edge in $A_v$ is to be explored, choose the first unexplored edge in $A_v$. A simple recursive procedure implementing this technique is presented in [ 10 ], along with proofs of the properties of depth-first search. Such a search requires $O(|V| + |E|)$ time, if the graph has $|V|$ vertices and $|E|$ edges. Properties necessary to this paper are described below.

Suppose G is a connected, undirected multigraph. A search of G imposes a direction on each edge of G given by the direction in which the edge is traversed when the search is performed. Thus G is converted into a directed graph G'. If the search is depth-first, then G' has special properties, given by the following lemma:

Lemma 9: Let P be the directed multigraph generated by a depth-first search of a connected multigraph G. Then P

is a palm tree. That is, P contains two disjoint sets of edges, one set denoted by $\rightarrow$ and the other set denoted by $\dashrightarrow$, such that the edges $\rightarrow$ determine a spanning tree of P and if $v \dashrightarrow u$ then $u \overset{*}{\rightarrow} v$ (u is an ancestor of v in the spanning tree).

A proof of Lemma 9 appears in [ 10 ]. Since a palm tree has no edges interconnecting the paths in its spanning tree, depth-first search is useful in solving connectivity problems. Figure 3 illustrates the palm tree obtained by a depth-first search of the graph G illustrated in Figure 1. Algorithms based on depth-first search for determining the biconnected components of an undirected graph and for determining the strongly connected components of a directed graph are given in [10]. In the next sections we extend the ideas in [ 10 ] to give an algorithm for determining the triconnected components of an undirected graph.

Finding Separation Pairs

Let $G = (V,E)$ be a biconnected multigraph with $|V|$ vertices and $|E|$ edges. The main problem in dividing G into its triconnected components is finding its separation pairs. This section gives a simple criterion, based upon depth-first search, for identifying the separation pairs of a multigraph. Two depth-first searches and some auxiliary calculations must be carried out. These calculations form the first part of the triconnected components algorithm and are outlined below. The definitions for the quantities LOWPT1, ND, etc., used in the outline will be given subsequently.

1. Perform a depth-first search on the multigraph G, converting G into a palm tree P. Number the vertices of G in the order they are reached during the search. Calculate LOWPT1(v), LOWPT2(v), and ND(v) for each vertex v in P.

2. Construct an acceptable adjacency structure A for P by ordering the edges in an adjacency structure according to the LOWPT1 and LOWPT2 values.

3. Perform a depth-first search of P using the adjacency structure A. Renumber the vertices of A from v to 1 in the order they are last examined during the search. Recalculate LOWPT1(v) and LOWPT2(v) using the new vertex numbers. Calculate SON1(v), DEGREE(v), and HIGHPT(v) for each vertex v.

The details of these calculations appear below. From Steps 1, 2, and 3 we get enough information to rapidly determine the separation pairs of G. Lemma 17 gives a condition for this purpose.

Suppose G is explored in a depth-first manner, giving a palm tree P. Let the vertices of P be numbered from 1 to $|V|$ so that $v \overset{*}{\rightarrow} w$ in P implies $v < w$, if we identify vertices by their number. For any vertex v in P, let ND(v) be the number of proper descendants of v. Let $\text{LOWPT1}(v) = \min(\{v\} \cup \{w \mid v \overset{*}{\rightarrow} \dashrightarrow w\})$. That is, LOWPT1(v) is the lowest vertex reachable from v by traversing zero or more tree arcs in P followed by at most one frond. Let $\text{LOWPT2}(v) = \min[\{v\} \cup (\{w \mid v \overset{*}{\rightarrow} \dashrightarrow w\} - \{\text{LOWPT1}(v)\})]$. That is, LOWPT2(v) is the <u>second</u> lowest vertex reachable from v by traversing zero or more tree arcs followed by at most one frond of P, if such a vertex exists. Otherwise, (i.e. if LOWPT1(v) = v), LOWPT2(v) = v also.

<u>Lemma 10</u>: $\text{LOWPT1}(v) \overset{*}{\rightarrow} v$ and $\text{LOWPT2}(v) \overset{*}{\rightarrow} v$ in P.

<u>Proof</u>: $\text{LOWPT1}(v) \leq v$ by definition. If LOWPT1(v) = v the result is immediate. If LOWPT1(v) < v there is a frond $u \dashrightarrow \text{LOWPT1}(v)$ such that $v \overset{*}{\rightarrow} u$. Since $u \dashrightarrow \text{LOWPT1}(v)$ is a frond, $\text{LOWPT1}(v) \overset{*}{\rightarrow} u$. Since P is a tree, $v \overset{*}{\rightarrow} u$ and $\text{LOWPT1}(v) \overset{*}{\rightarrow} u$, either $v \overset{*}{\rightarrow} \text{LOWPT1}(v)$ or $\text{LOWPT1}(v) \overset{*}{\rightarrow} v$. But LOWPT1(v) < v. Thus it must be the case that $\text{LOWPT1}(v) \overset{*}{\rightarrow} v \overset{*}{\rightarrow} u$, and the lemma holds for LOWPT1(v). The proof is the same for LOWPT2(v).

Lemma 11: Suppose LOWPT1(v) and LOWPT2(v) are defined relative to some numbering which satisfies $v \overset{*}{\rightarrow} w$ in P implies NUMBER(v) < NUMBER(w). Then LOWPT1(v) and LOWPT2(v) identify unique vertices independent of the numbering used.

Proof: LOWPT1(v) always identifies an ancestor of vertex v. Furthermore LOWPT1(v) is the lowest numbered ancestor of v with a certain property relating to the palm tree P. Since the order of the ancestors of v corresponds to the order of their numbers, LOWPT1(v) identifies a unique vertex independent of the numbering; namely, the first ancestor of v along the path $1 \overset{*}{\rightarrow} v$ which has the desired property. (Any satisfactory numbering assigns 1 to the root of P.) The proof is the same for LOWPT2(v).

Step 1 of the calculations may be carried out in $O(|V| + |E|)$ time using an adjacency structure for the depth-first search. A program for this purpose appears below. Numbering the vertices in the order they are reached during the search clearly guarantees that $v \overset{*}{\rightarrow} w$ implies $v < w$.

```
STEP1: begin
           integer i;
           procedure SEARCH1 (v,u); comment vertex u is the
                     father of vertex v in the spanning tree
                     being constructed;
       begin
          ND(v):= 0;
```

```
A:      LOWPT2(v):= NUMBER(v):= i:= i+1
        for w in the adjacency list of v do
          begin
            if w is not yet numbered then
              begin
                construct tree arc v → w in P;
                FATHER(w):= v;
B:              SEARCH1 (w,v);
                ND(v) = ND(v) + ND(w);
C:              if LOWPT1(w) < LOWPT1(v) then
                  begin
D:                  LOWPT2(v):= min(LOWPT1(v),LOWPT2(w));
                    LOWPT1(v):=LOWPT1(w);
                  end
                else if LOWPT1(w) = LOWPT1(v) then
E:                  LOWPT2(v):= min(LOWPT2(v),LOWPT2(w))
F:              else LOWPT2(v):= min(LOWPT2(v),LOWPT1(w));
              end
            else if (NUMBER(w) < NUMBER(v)) and (w¬=u)
              or (FLAG(v)+FALSE)) then
              begin
                construct frond v -→w in P;
G:              if NUMBER(w) < LOWPT1(v) then
                  begin
                    LOWPT2(v):= LOWPT1(v);
                    LOWPT1(v):= NUMBER(w);
                  end
```

```
                else if NUMBER(w) > LOWPT1(v) then
H:                      LOWPT2(v):= min(LOWPT2(v), NUMBER(w));
              end
            else if (w=a)and(FLAG(v)=TRUE) then FLAG(v):=FALSE;
          end;
      end;
I: i:=0;
    for v:=1 until V do FLAG(v):=TRUE; comment FLAG(v) becomes
      false when edge u→v is traversed backwards for the first time;
J:  SEARCH1 (s,0); comment s is an arbitrary starting vertex;
  end;
```

Intuitively, STEP1 works as follows: Initially variable $i$ is set equal to 0(statement I). The depth-first search begins at the root of the palm tree $P$ to be constructed (statement J). Each time a new vertex is reached, recursive procedure SEARCH1 is called, to continue the search starting at the new vertex. A vertex is numbered the first time it is reached (statement A). The value af LOWPT1(v) is set to NUMBER(v) the first time $v$ is reached (statement A). If $v \rightarrow w$, then on completion of the search of the subtree $T_w$ containing the descendants of $w$, LOWPT1(v) is set to LOWPT1(w) if LOWPT1(w) is less than the current value of LOWPT1(v) (statement C). If a frond $v \dashrightarrow w$ is encountered and NUMBER(w) is less than the current value of LOWPT1(v), then LOWPT1(v) is set to NUMBER(w) (statement G). Thus, after $w$ is examined for the last time,

$$\text{LOWPT1}(v) = \min(\{v\} \cup \{\text{LOWPT1}(w) \mid v \rightarrow w\} \cup \{w \mid v \dashrightarrow w\}) .$$

It is easy to show by induction on the order in which vertices

are last examined that, for each $v$,

$$\text{LOWPT1}(v) = \min(\{v\} \cup \{w \mid v \overset{*}{\rightarrow} w\}) .$$

The computation of LOWPT2 is similar to that of LOWPT1. The value of LOWPT2(v) is set to NUMBER(v) the first time $v$ is reached (statement A). If $v \rightarrow w$, then on completion of the search of the subtree $T_w$ containing the descendants of $w$, LOWPT2(v) is modified as follows:

Since $T_w$ is completely explored, LOWPT1(w) and LOWPT2(w) are completely calculated. We must update the value of LOWPT2(v) knowing the current values of LOWPT1(v) and LOWPT2(v). If LOWPT1(w) $<$ LOWPT1(v), then min(LOWPT2(w),LOWPT1(v)) is the value of LOWPT2(v) (statement D). If LOWPT1(w) = LOWPT2(v), then min(LOWPT2(w),LOWPT2(v)) is the new value of LOWPT2(v) (statement E). If LOWPT1(w) > LOWPT2(v), then min(LOWPT1(w), LOWPT2(v)) is the new value of LOWPT2(v) (statement F). If a frond $v \rightarrow w$ is encountered, we must update the value of LOWPT2(v) similarly. If NUMBER(w) < LOWPT1(v), the current value of LOWPT1(v) is the new value of LOWPT2(v) (statement G). If NUMBER(w) = LOWPT1(v), LOWPT2(v) is unchanged. If NUMBER(w) > LOWPT1(v), min(NUMBER(w),LOWPT2(v)) is the new value of LOWPT2(v) (statement H). We may show by induction that LOWPT2 is calculated correctly.

Let $\phi$ be the mapping from the edges of $P$ into $\{1,2,\ldots,2|V|+1\}$ defined by:

(i) If $e = v \rightarrow w$, $\phi(e) = 2w + 1$.

(ii) If $e = v \rightarrow w$ and $LOWPT2(w) < v$, $\phi(e) = 2LOWPT1(w)$.

(iii) If $e = v \rightarrow w$ and $LOWPT2(w) \geq v$, $\phi(e) = 2LOWPT1(w)+1$.

Let A be an adjacency structure for P. A is called acceptable if the edges e in each adjacency list of A are ordered according to increasing value of $\phi(e)$.

Lemma 12: Let P be a palm tree of a biconnected graph G whose vertices are numbered so that $v \overset{*}{\rightarrow} w$ in P implies $v < w$. Then the acceptable adjacency structures of P are independent of the exact numbering scheme.

Proof: If $v \rightarrow w$ in P, then by Lemma 10, LOWPT2(w) is an ancestor of w. By Lemma 11, LOWPT2(w) is a fixed vertex independent of the numbering. Since the order of the ancestors is independent of the numbering, the question as to whether LOWPT2(w) is less than v is independent of the numbering. Since G is biconnected if $v \rightarrow w$ in P, then $LOWPT1(w) \leq v$ by Lemma 5 of [10]. By Lemma 10, LOWPT1(w) is an ancestor of w. Since $LOWPT1(w) \leq v$, LOWPT1(w) must be an ancestor of v. By Lemma 11, the vertex corresponding to LOWPT1(w) is independent of the numbering scheme. Similarly if $v \rightarrow w$, then by Lemma 9 w is an ancestor of v. But the order of the ancestors of v is identical to the order of their numbers, and this order is independent of the numbering. Thus the acceptable adjacency structures A for P depend only on P and not on the exact numbering.

In general, a palm tree P has many acceptable adjacency structures. Given a satisfactory numbering of the vertices

of P, we may easily construct an acceptable adjacency structure A by using a radix sort with $2|V| + 1$ buckets. The following procedure gives the sorting algorithm, which is Step 2 of the calculations. All vertices are identified by number. It is obvious that the sorting procedure requires $O(|V| + |E|)$ time.

```
procedure SORT;
  begin
    for each arc (v,w) of P do
      if v -→ w then place (v,w) in BUCKET(2*w+1)
      else if LOWPT2(w) < v then place (v,w) in
          BUCKET(2*LOWPT1(w))
      else place (v,w) in BUCKET(2*LOWPT1(w)+1);
    for i:= 1 until v do construct empty adjacency list
      for vertex v;
    for i:= 1 until 2*v+1 do
      for each arc (v,w) in BUCKET(i) do
        place w at end of adjacency list of vertex v;
  end;
```

In Step 3 of the calculations we perform a depth-first search of P using the acceptable adjacency structure A given by Step 2. During Step 3 we calculate certain values necessary to determine the separation pairs of G. Let the vertices of P be numbered so that $v \overset{*}{\to} w$ implies $v < w$. If $u \to v$ in P, let $HIGHPT(v) = \max(\{v\} \cup \{w | v \overset{*}{\to} w \to u\})$.

HIGHPT(v) is the highest numbered vertex which is the tail of a frond whose head is the father of v, if such a vertex exists. (Otherwise HIGHPT(v) = v). SON1(v) is the first son of v reached during the search. DEGREE(v) is the number of edges incident to vertex v.

During Step 3 we number the vertices from v to 1 in the order they are last examined during the search. It is clear that this numbering scheme guarantees that $v \overset{*}{\rightarrow} w$ implies $v < w$. We also calculate LOWPT1(v), LOWPT2(v), SON1(v), and HIGHPT(v) with respect to the new numbering scheme. A program to implement Step 3 appears below. It is easy to verify that the numbering and calculations are carried out correctly.

```
STEP3: begin
          integer i;
          procedure SEARCH2(v);
             begin
             A: NUMBER(v):= i-ND(v);
                LOWPT1(v):= LOWPT2(v):= HIGHPT(v):= NUMBER(v);
                for w in the adjacency list A_v of v do
                   if w is not yet numbered then
                      begin
                         SEARCH2(w);
             B: if SON1(v) = 0 then SON1(v)1 = NUMBER(w);
                i:= i-1;
```

```
                if  LOWPT1(w) < LOWPT1(v) then
                    begin
                        LOWPT2(v):= min (LOWPT1(v), LOWPT2(w));
                        LOWPT1(v):= LOWPT1(w);
                    end
                else if  LOWPT1(w) = LOWPT1(v)  then
                    LOWPT2(v):= min (LOWPT2(v), LOWPT2(w))
                else  LOWPT2(v):= min (LOWPT2(v), LOWPT1(w));
            end
        else
            begin
                HIGHPT (FATHER(w)) = max (HIGHPT(FATHER(w)),
                NUMBER(V));
                if  NUMBER(w) < LOWPT1(v)  then
                    begin
                        LOWPT2(v):= LOWPT1(v);
                        LOWPT1(v):= NUMBER(w);
                    end
                 else if  NUMBER(w) > LOWPT1(v)  then
                    LOWPT2(v):= min (LOWPT2(v), NUMBER(w));
            end
        end;
    for i= 1 to (V)  do  SON1(v):= 0;
C:  i:= V;
    SEARCH2(r); comment  r  is the root of  P;
end
```

Step 3 numbers the vertices from $|V|$ to 1 in the order they are last reached during the search. However, each vertex must actually be assigned a number the first time it is reached, in order for the calculations of LOWPT1, LOWPT2, and HIGHPT to proceed correctly. In order to accomplish this, variable i is set equal to $|V|$ when the search begins (statement C). The value of i is decreased by one each time a new vertex is discovered (statement B). Thus when a vertex v is first reached, i is equal to the number we want to assign to v minus the number of vertices to be examined before v is examined for the last time. But the vertices to be reached between the time v is first examined and the time v is last examined are just the proper descendants of v. Thus if we assign the number i- ND(v) to v when v is first examined (statement A), the numbering will be correct. The other calculations performed in Step 3 are straightforward. Step 3 also requires $O(|V| + |E|)$ time. The palm tree for the graph G of Figure 1 is illustrated in Figure 4 along with LOWPT1 and LOWPT2 values.

Let G be a biconnected multigraph on which Steps 1, 2, and 3 have been performed, giving a palm tree P and the sets of values defined above. Let A with adjacency lists $A_v$ be the acceptable adjacency structure constructed in Step 2. Let the vertices of G be identified by the numbers assigned in Step 3. We need one more definition. If $u \rightarrow v$ and v is the first entry in $A_u$, then v is called a first son of $A_u$. (For each vertex v, SON1(v), the first son of v, is calculated in Step 3.) If $u_0 \rightarrow u_1 \rightarrow \cdots \rightarrow u_n$ and $u_i$

is a first son of $u_{i-1}$ for $1 \leq i \leq n$, then $u_n$ is called a first descendant of $v_0$. The lemmas below give the properties we need to determine the separation pairs of G.

Lemma 13: Let $A_u$ be the adjacency list of vertex u. Let $u \rightarrow v$ and $u \rightarrow w$ be tree arcs with v occurring before w in $A_u$. Then $u < w < v$.

Proof: Step 3 numbers the vertices from v to 1 in the order they are last examined in the search. If $u \rightarrow v$ is explored before $u \rightarrow w$, v will be examined last before w is examined last, and v will receive a higher number. Clearly u will be last examined after both v and w are last examined, so u receives the smallest number of the three vertices.

Lemma 14: A is acceptable with respect to the numbering given by Step 3.

Proof: The sort in Step 2 creates an acceptable adjacency structure for the original numbering. By Lemma 13, $u \rightarrow v$ implies $u < v$ and hence by Lemma 12, A is acceptable for the new numbering.

Lemma 15: If v is a vertex and D(v) is the set of descendants of v, then $D(v) = \{x | v \leq x \leq v + ND(v)\}$. If w is a first descendant of v, then $D(v) - D(w) = \{x | v \leq x < w\}$.

Proof: Consider searching P in reverse of the order used in Step 3. Vertices will be examined for the first time in ascending order from 1 to $|v|$. Consequently descendants

of $v$ must be assigned consecutive numbers from $v$ to $v + ND(v)$. If $w$ is a first descendant of $v$, vertices in $D(w)$ will be assigned numbers after all vertices in $D(v) - D(w)$. Thus $D(v) - D(w) = \{x | v \leq x < w\}$.

Lemma 16: Let $\{a,b\}$ be a separation pair in $G$ with $a < b$. Then $a \overset{*}{\not\to} b$ in the spanning tree $T$ of $P$.

Proof: Since $a < b$, $a$ cannot be a descendant of $b$. Suppose that $b$ is not a descendant of $a$ in T. Then there are no edges connecting $\wedge$ and $b$ in $P$. Let $E_i$, $1 \leq i \leq k$, be the separation classes with respect to $\{a,b\}$. Let $S = V - \{v | a \overset{*}{\to} v\} - \{w | b \overset{*}{\to} w\}$. The vertices $S$ define a subtree in $T$, not containing $a$ or $b$, so $E(S)$ must be contained in some separation class, say $E_1$. Let $c$ be any son of $a$. $E(D(c))$ must be contained in some separation class, say $E_2$. But since $G$ is biconnected, $LOWPT1(c) < a$, by Lemma 5 of [ 10 ]. Thus some edge is incident both to a vertex in $S$ and to a vertex in $D(c)$, and $E_1 = E_2$. A similar argument shows that edges incident to any descendant of $b$ are in $E_1$. But this means that $E_1 = E$, and $\{a,b\}$ cannot be a separation pair.

Lemma 17: Suppose $a < b$. Then $\{a,b\}$ is a separation pair of $G$ if and only if either (1), (2), or (3) below holds.

(1) There are distinct vertices $r \neq a,b$ and $s \neq a,b$ such that $b \to r$, $LOWPT1(r) = a$, $LOWPT2(r) \geq b$, and $s$ is not a descendant of $r$. (Pair $\{a,b\}$

is called a type 1 separation pair. The type 1 pairs for the graph in Figure 4 are (1,4),(1,5),(4,5) and (1,8).)

(2) There is a vertex $r \neq b$ such that $a \rightarrow r \overset{*}{\rightarrow} b$; $b$ is a first descendant of $r$; $a \neq 1$; every frond $x \dashrightarrow y$ with $r \leq x < b$ has $a \leq y$; and every frond $x \dashrightarrow y$ with $a < y < b$ and $b \rightarrow w \overset{*}{\rightarrow} x$ has $LOWPT1(w) \geq a$. (Pair $\{a,b\}$ is called a type 2 separation pair. The type 2 pairs for the graph in Figure 4 are (4,5) and (8,12)).

(3) $(a,b)$ is a multiple edge of $G$ and $G$ contains at least four edges.

Proof: The converse part of the lemma is easiest to prove. Suppose pair $\{a,b\}$ satisfies (1), (2), or (3). Let $E_i$, $1 \leq i \leq k$, be the separation classes of $G$ with respect to $\{a,b\}$. Suppose $\{a,b\}$ satisfies (1). Then the edge $(b,r)$ is contained in some separation class, say $E_1$. Every tree arc with an endpoint in $D(r)$ has e other endpoint in $D(r) \cup \{a,b\}$. Also, since $LOWPT1(r) = a$ and $LOWPT2(r) \geq b$, every frond with an endpoint in $D(r)$ has the other endpoint in $D(r) \cup \{a,b\}$. Thus $E_1$ consists of all edges with an endpoint in $D_r$. No other edges are in $E_1$ and the edges incident to vertex $s$ must be in some other class, say $E_2$. Since $E_1$ and $E_2$ each contain two or more edges, $\{a,b\}$ is a separation pair.

Suppose $\{a,b\}$ satisfies (2). Let $S = D(r) - D(b)$. All edges incident to a vertex in $S$ are in the same separation class, say $E_1$. Since $b$ is a first descendant

of r, $S = \{x | r \leq x < b\}$, by Lemma 15. Let $b_1, b_2, \ldots, b_n$ be the sons of b in the order they occur in $A_b$. Let $i_0 = \min \{i | \text{LOWPT1}(b_i) \geq a\}$. Since A is acceptable, $i < i_0$ implies $\text{LOWPT1}(b_i) < a$, and $i \geq i_0$ implies $\text{LOWPT1}(b_i) \geq a$. By (2), every frond with tail in S has its head in $S \cup \{a\}$. Also by (2), every frond with head in S has its tail in $S \cup \{b\} \cup (\bigcup_{i \geq i_0} D(b_i))$. Every edge with an endpoint in $D(b_i)$, $i \geq i_0$, has its other endpoint in $S \cup \{a,b\} \cup D(b_i)$. Thus the class $E_1$ contains <u>at least</u> all edges with an endpoint in S, and <u>at most</u> all edges with an endpoint in $S \cup \bigcup_{i \geq i_0} D(b_i)$.

Since $a \neq 1$, the edges incident to the root of P cannot be in $E_1$, and therefore $\{a,b\}$ is a separation pair.

Suppose $\{a,b\}$ satisfies (3). Then it is clear that $\{a,b\}$ is a separation pair.

We now must prove the direct part of the lemma. Thus suppose that $\{a,b\}$ is a separation pair with $a < b$. If $(a,b)$ is a multiple edge of $G$ then it is clear that $\{a,b\}$ satisfies (3). Thus suppose that $(a,b)$ is not a multiple edge of $G$. By Lemma 16, $a \overset{*}{\rightarrow} b$. Let $E_i$, $1 \leq i \leq k$, be the separation classes of $G^-$ with respect to $\{a,b\}$. Let $a \rightarrow v \overset{*}{\rightarrow} b$. Let $S = D(v) - D(b)$. Let $X = V - D(a)$. (Either $S$ or $X$ or both may be empty.) $E(S)$ and $E(X)$ are each contained in a separation class, say $E(S) \subseteq E_1$ and $E(X) \subseteq E_2$.

Let $a_i \neq v$ be a son of $a$. If $a$ has such a son, $LOWPT1(a_i) < a$. This means that $E(D(a_i)) \subseteq E_2$. Let $Y = X \cup_{a_i} \cup D(a_i)$. Let $b_1, b_2, \ldots, b_n$ be the sons of $b$ in the order they occur on the adjacency list of $b$. Let $E(D(b_i))$ be the set of edges with an endpoint in $D(b_i)$. The separation classes must be unions of the sets $E(S)$, $E(Y)$, $\{(a,b)\}$, $E(D(b_i))$, $E(D(b_2))$, ..., $E(D(b_n))$.

If $E(D(b_i)) = E_j$ for some $i$ and $j$, then $LOWPT1(b_i) = a$, since $G$ is biconnected and this means $LOWPT1(b_i) < b$ by Lemma 5 of [ 10 ]. Also, $LOWPT2(b_i) \geq b$. Since $\{a,b\}$ is a separation pair, there must be a separation class other than $E_j$ and $\{(a,b)\}$. Thus there is a vertex $s$ such that $s \neq a$, $s \neq b$, and $s \notin D(b_i)$. This means that $\{a,b\}$ satisfies (1) where $r$ is $b_i$.

Suppose now that no $E(D(b_i))$ is by itself a separation class. Let $i_0 = \min\{i \mid LOWPT1(b_i) \geq a\}$. If $i \geq i_0$ then since G is biconnected it must be the case that $LOWPT2(b_i) \leq b$, and the separation classes are $E_1 = E(S) \cup \bigcup_{i \geq i_0} E(D(b_i))$, $E_2 = E(Y) \cup \bigcup_{i<i_0} E(D(b_i))$, $E_3 = \{(a,b)\}$. ($E_3$ may be empty.) We have $v \neq b$ since $\{a,b\}$ is not a type 1 pair and $a \neq 1$ since $E_2$ is non-empty. If $x \rightarrow y$ is a frond with $v \leq x < b$, then $x \in S$, $(x,y) \in E_1$, and $a \leq y$. If $x \rightarrow y$ is a frond with $a < y < b$ and $b \rightarrow b_i \overset{*}{\rightarrow} x$, then $y \in S$ $(x,y) \in E_1$, and $i \geq i_0$, which means that $LOWPT1(b_i) \geq a$. We must verify one more condition to show that (2) holds; namely, that b is a first descendant of v. Since G is biconnected, LOWPT1(v) < a. Thus some frond with tail in D(v) has head less than a. By the construction of A and the definition of a first descendant, there exists some frond $x \rightarrow y$ with $x \in D(v)$ and $y < a$ such that x is a <u>first</u> descendant of v. If b were not a first descendant of v then x would be in S, and $E_1$ and $E_2$ could not be distinct separation classes. Thus b is a first descendant of v, and (2) holds with r = v. This completes the proof of the direct part of the lemma.

Lemma 17 and its proof are worth pondering carefully. The lemma gives three easy-to-apply conditions for separation pairs. Conditions (1) and (2) identify the non-trivial separation pairs of the multigraph. Condition (3) handles multiple edges. Condition (1) requires that a simple test be performed

on each tree arc of P. Thus testing for type 1 pairs requires $O(|V|)$ time. Testing for type (2) pairs is somewhat harder but may be done in $(|V|+|E|)$ time using another depth-first search. If $\{a,b\}$ is a type 2 pair, $a \rightarrow v \overset{*}{\rightarrow} b$, and $i_0 = \min\{i|\text{LOWPT1}(b_i) \geq a\}$, where $b_1, b_2, \ldots, b_n$ are the sons of $b$ in the order they occur on the adjacency list of $b$, then one separation class with respect to $\{a,b\}$ is $E(\{x|v \leq x \leq b_{i_0} + \text{ND}(b_{i_0})\} - \{b\})$. This follows from Lemma 15 and the proof of Lemma 17. The numbering given by Step 3 thus makes it easy to determine the separation classes and to divide the graph once a separation pair is found. The complete triconnectivity algorithm is given in the next section.

## A Triconnectivity Algorithm

By using the simple criterion for separation pairs given in the last section, we may divide a graph into its triconnected components in $O(|V| + |E|)$ time. This is accomplished by carrying out another depth-first search of $G$, again using the acceptable adjacency structure $A$ constructed for $G$. We use Lemma 17 to find separation pairs. Each time an edge is backed over during the search it is added to a stack of edges. Each time a separation pair is found a set of edges corresponding to a split component is removed from the stack. A virtual edge is added to the component and a corresponding virtual edge is added to the stack. The complete search gives a set of split components for $G$. Assembling the split components to give the triconnected components of $G$ is then a simple matter.

Suppose $\{a,b\}$ is a separation pair with $a \overset{*}{\rightarrow} b$. If $\{a,b\}$ is a pair of type 1, Lemma 17 enables us to detect this fact when vertex $b$ is examined during the search. The corresponding split component may be deleted and replaced with a virtual edge $(a,b,i)$. If $\{a,b\}$ is a multiple edge, this fact is easy to detect and calculation of the corresponding split component is easy. Similarly, if $a \rightarrow v \rightarrow b$ and $v$ is a vertex of degree 2, detection of this fact is easy. Finding separation points of type 2 is a little more difficult than the other cases, however. The procedure for finding type 2 pairs will be outlined; then the entire algorithm will be

presented, followed by a proof of its correctness and time bound.

We keep a stack which contains triples of vertices. If $(h,a,b)$ is a triple on the stack, $\{a,b\}$ is a possible type 2 separation pair and $h$ is the highest numbered vertex in the corresponding split component. The stack containing these triples is updated in the following manner during the depth-first search. Initially the stack is empty. The stack is modified whenever we back over an edge.

Whenever we back over a frond $v \dashrightarrow w$, we delete all triples $(h,a,b)$ on top of the stack with $w < a$. If $(h_1,a,b_1)$ is the last triple deleted, we add a new triple $(h_1,w,b_1)$ to the stack. If no triples are deleted we add $(v,w,v)$ to the stack.

Whenever we return to a vertex $v \neq 1$ along a tree arc $v \rightarrow w$, we test the top triple $(h,a,b)$ on the stack to see if $v = a$. If so, $\{a,b\}$ is a type 2 pair. We delete all triples $(h,a,b)$ on top of the stack with $\text{HIGHPT}(w) > h$.

If $w$ is not the first son of $v$, we carry out the following steps. Let $w + \text{ND}(w)$ be the highest descendant of $w$. Delete all triples $(h,a,b)$ on top of the stack with $w + \text{ND}(w) \geq b$. Then we delete all triples with $\text{LOWPT1}(w) < a$. If no triples are deleted during the latter step and $\neg(\text{LOWPT1}(w) \rightarrow v)$, then we add the triple $(w + \text{ND}(w), \text{LOWPT1}(w), v)$ to the stack. Otherwise, if $(h,a,b)$ was the last triple deleted such that $\text{LOWPT1}(w) < a$, we add $(\max\{w + \text{ND}(w), h\}, \text{LOWPT1}(w), b)$ to the stack.

The reason for constructing the acceptable adjacency structure A using LOWPT2 as well as LOWPT1 is to make sure that fronds are handled correctly. Suppose v is a vertex in G, $v \rightarrow w_i$ for $1 \leq i \leq k$, with LOWPT1(i) = u, and $v \dashrightarrow u$. Further suppose that the $w_i$ are ordered as they appear in $A_v$, the adjacency list of v. Then there is some $i_0$ such that $i \leq i_0$ => LOWPT2(i) < v and $i > i_0$ = LOWPT2(i) $\geq$ v. In $A_v$, u will appear after all the $w_i$, $1 \leq i \leq i_0$. Each vertex $v_i$, $i > i_0$, is part of a split component for the separation pair {a,b}. When such a component is deleted from G, it will be replaced by a virtual edge (a,b,i) which is a frond. It is important that all the $w_i$, $i > i_0$, appear together in $A_v$, so that these virtual fronds may be located and combined to give split components which are bonds.

An Algol-like procedure to find the split components of a biconnected multigraph appears below. Steps 1, 2, and 3 described in the previous section must be carried out before the triconnectivity algorithm is applied.

```
begin
  integer j;
  procedure TRICON(v);
        for w in the adjacency list of v do
            if v → w then
                begin
                    TRICON(w);
```

```
Comment program has just returned along
    tree arc v → w;
add v → w to EDGESTACK;

Comment test for type 1 component;
if ((LOWPT2(w) ≥ v) & ((LOWPT1(w) ≠ 1) |
   (DEGREE(v) > 2) |(FATHER(v) ≠ LOWPT1(v)))
    then
    begin
        j:= j+1
        while (x,y) on top of EDGESTACK
            has x ≥ w do
            begin
                delete (x,y) from EDGESTACK;
                add (x,y) to new component;
                decrement DEGREE(x),DEGREE(y);
            end;
        add(v,LOWPT1(w) ,j) to new component;
        output new component;
        comment test for multiple edge;
        if (x,y) on top of EDGESTACK satisfies
            (x,y) = (v,LOWPT1(w)) then
            begin
                j:= j+1;
                add (x,y),(v,LOWPT1(w),j-1),
                    (v,LOWPT1(w),j) to new
                    component;
```

```
                output new component;
                decrement DEGREE(v),
                    DEGREE(LOWPT1(v));
            end;
        add(v,LOWPT1(w),j) to EDGESTACK;
        increment DEGREE(v),DEGREE(LOWPT1(w));
    end;

comment test for degree 2 vertex (component
    will be of type 2);
if (DEGREE(w) = 2) & ((v ≠ 1)|(second edge
    (w,x) on EDGESTACK has DEGREE(x) > 2)) then
    begin
        j:= j+1
        add top two edges (v,w) and (w,x) on
            EDGESTACK to new component;
        if (y,z) on top of EDGESTACK has
            (y,z) = (x,v) then
            begin
                j:= j+1
                add(y,z),(x,v,j-1),(x,v,j) to
                    new component;
                output new component;
                decrement DEGREE(v),DEGREE(x);
            end;
        add (x,v,j) to EDGESTACK;
        FATHER(x):= v;
        if son1(v) = w then son1(v):= x;
    end;
```

```
comment test for type 2 pair;
while (h,a,b) on top of TRIPLESTACK has
    a = FATHER(b) do delete (h,a,b) from
    TRIPLESTACK;
while (h,a,b) on top of TRIPLESTACK has
    (v = a) & (a ≠ 1) do
    begin
        j:= j+1;
        delete (h,a,b) from TRIPLESTACK;
        while (x,y) on top of EDGESTACK has
            (a ≤ x ≤ h) & (a ≤ y ≤ h) do
            if (x,y) = (a,b) then
            begin
                FLAG:= true;
                save edge (x,y);
            end
        else
            begin
                delete (x,y) from EDGESTACK
                    and add to new component;
                decrement DEGREE(x),DEGREE(y)
            end;
        add (a,b,j) to new component;
        if FLAG = true then
            begin
                FLAG:= false;
                j:= j+1
                add (x,y),(a,b,j-1),(a,b,j)
                    to new component;
```

```
                output new component;
            end;
        end;

    add (a,b,j) to EDGESTACK;
    increment DEGREE(a),DEGREE(b);
    comment fix up TRIPLESTACK;
    while (h,a,b) on TRIPLESTACK has
        HIGHPT(w) > h do
        delete (h,a,b) from TRIPLESTACK;
    if w ≠ SON1(v) then
        begin
            while (h,a,b) on TRIPLESTACK has
                w + ND(w) ≥ b do
                delete (h,a,b) from TRIPLESTACK
            while (h,a,b) on TRIPLESTACK has
                LOWPT1(w) < a do
                begin
                    FLAG:= true;
                    delete (h,a,b) from
                        TRIPLESTACK;
                end;
            if ¬FLAG & ¬(LOWPT1(w) = FATHER(v)) then
                add (w + ND(w),LOWPT1(w),v) to
                    TRIPLESTACK
            else if FLAG then
                begin
                    FLAG:= false;
```

```
                        if (h,a,b) last triple deleted
                            from TRIPLESTACK and
                            FATHER(b) ≠ LOWPT1(w)
                            then add (max{w+ND(w),h},
                            LOWPT1(w),b) to
                            TRIPLESTACK;
                    end;
            end;
    end
else
    begin
        comment v -→ w;
        comment test for multiple edge;
        if w = FATHER(v) then
            begin
                j:= j+1;
                add v → w, v -→ w,  (v,w,j) to
                    new component;
                mark tree arc v → w as virtual edge j;
                decrement DEGREE(v),DEGREE(w);
            end
        else if (x,y) on EDGESTACK has (x,y) = (v,w)
            then
            begin
                j:= j+1;
                add (x,y),(v,w),(v,w,j) to new
                    component;
```

```
                    add (v,w,j) to EDGESTACK;
                    decrement DEGREE(v),DEGREE(w);
                end
            else add (v,w) to EDGESTACK;
            comment fix up TRIPLESTACK;
            while (h,a,b) on TRIPLESTACK has
                begin
                    FLAG:= true
                    delete (h,a,b) from TRIPLESTACK;
                end;
            if FLAG = true then
                begin
                    FLAG:= false
                    if (h,a,b) is last triple deleted
                        from TRIPLESTACK then add
                        (h,w,b) to triplestack;
                end
            else add (v,w,v) to TRIPLESTACK;
        end;
  j:= 0;
  TRICON(1); comment vertex 1 is starting vertex for search;
end;
```

Theorem 18: The algorithm above correctly divides a biconnected multigraph into split components.

Proof: Unfortunately, the algorithm looks much more complicated than it is. The ideas in it are a direct application of Lemma 17. We must prove two things: (1) if the graph is triconnected, the algorithm will not split it; (2) if the graph is not triconnected, the algorithm will split it. Once we have these two facts, we may prove the Theorem by induction on the number of edges in the graph.

The tests for multiple edges, for type 1 separation pairs, and for degree 2 vertices are straightforward. These tests will discover a separation pair of the correct type if one exists, and they will not report a separation pair if one does not exist. Thus we must only show that the type 2 test works correctly on multigraphs with no degree 2 vertices, and we will have verified (1) and (2).

Suppose $G$ is a multigraph with no degree 2 vertices, no multiple edges, and no type 1 pairs. Consider the type 2 test and the changing contents of TRIPLESTACK as the search of $G$ progresses.

If $(h_1,a_1,b_1)$ occurs above $(h_2,a_2,b_2)$ on the stack, $a_2 \leq a_1$ and if $a_2 = a_1$ then $b_2 \leq b_1$. Further, if $(h,a,b)$ is deleted from the stack because a frond $v \dashrightarrow w$ is found with $w < a$, then $v < b$. These facts may be proved by induction on the search step using the ordering given by the adjacency structure $A$. The crucial fact to notice is that if $(h,a,b)$ is deleted from the stack because frond $v \dashrightarrow w$ with $w < a$ is found, then $v \geq b$ implies $b \overset{*}{\rightarrow} v$ by the numbering scheme. The triple $(h,a,b)$

corresponds to some frond $x \dashrightarrow y$ with $a \leq x \leq b$ and $a \leq y \leq b$. But if $b \overset{*}{\rightarrow} v$ and $w < a$, the frond $v \dashrightarrow w$ should have been traversed before the frond $x \dashrightarrow y$, because of the ordering in A. This contradiction establishes that $v < b$. The first statement above follows by examining each step during which the algorithm adds or deletes a triple from the stack.

Every triple (h,a,b) on the stack also satisfies $a \overset{*}{\rightarrow} b$ and $a \overset{*}{\rightarrow} v$, $v \neq a$, where v is the vertex currently being examined during the search. This follows again by examination of the steps which add and delete triples.

If triple (h,a,b) on the stack is tested and it is found that $v = a \neq 1$ when returning along a tree arc $v \rightarrow w$, it is straightforward to prove by induction on the search step that {a,b} is a type 2 separation pair. We merely verify that if any of the conditions in Lemma 17 are false, (h,a,b) must previously have been deleted from the stack. Conversely, suppose G has a type 2 pair (a,b). Let $b_1,\ldots,b_n$ be the sons of b in the order they occur in $A_b$. Let $i_0 = \min \{i \mid \text{LOWPT1}(b_i) \geq a\}$. Let $h = b_{i_0} + \text{ND}(b_{i_0})$. Finally, suppose $a \rightarrow v \overset{*}{\rightarrow} b$ and that $i \dashrightarrow j$ is the first frond traversed during the search with $v \leq i \leq h$. Then we may prove by induction on the search step that (i,j,i) is placed on the stack, possibly modified, and eventually is selected as a type 2 pair. Thus the type 2 test works correctly, and the

algorithm splits a multigraph if and only if a separation pair exists.

The theorem follows by induction on the number of edges in $G$. Suppose the theorem is true for graphs with fewer than $k$ edges. Let $G$ have $k$ edges. If $G$ cannot be split, the algorithm works correctly on $G$ by the discussion above. If $G$ can be split, it will be split. Consider the first split performed by the algorithm, producing split graphs $G_1$ and $G_2$. The behavior of the algorithm on $G$ is a composite of its behavior on $G_1$ and $G_2$. Since the algorithm splits $G_1$ and $G_2$ correctly by the induction hypothesis, it must split $G$ correctly. The Theorem follows by induction.

Figure 5 gives the contents of EDGESTACK and TRIPLESTACK when the first biarticulation point pair (8,12) is detected.

Theorem 19: The algorithm for finding a set of split components for a biconnected multigraph $G$ with $|V|$ vertices and $|E|$ edges requires $O(|V| + |E|)$ time and space. The space required by the algorithm is obviously $O(|V|+|E|)$. The preliminary calculations require $O(|V|+|E|)$ time as discussed in the previous section.

Proof: The number of edges in a set of split components of $G$ is bounded by $3|E| - 6$ by Lemma 1. Each such edge is placed on the edge stack once and deleted once during the search. The search itself requires $O(|V|+|E|)$ time, including the various tests. The number of triples added to TRIPLESTACK is bounded by $|V|+|E|$. Each triple may

only be modified if it is on top of the stack. Thus the time for maintaining the stack of triples is also $O(|V| + |E|)$, and the entire algorithm requires $O(|V| + |E|)$ time.

Thus we have an $O(|V| + |E|)$ algorithm for determining the split components of a biconnected multigraph. Use of an $O(|V| + |E|)$ algorithm for finding the biconnected components of a multigraph [ 7,10 ] enables us to extend the algorithm above to arbitrary multigraphs. It is easy to devise an $O(|V| + |E|)$ algorithm to merge the triangles and bonds. If $\mathcal{S}_1$ is the set of triangles and $\mathcal{S}_2$ is the set of bonds among the split components, we merely construct the auxillary graphs $S(\mathcal{S}_1)$ and $S(\mathcal{S}_2)$ and look at their connected components. Thus we have an $O(|V| + |E|)$ algorithm for determining the triconnected components of an arbitrary multigraph. Such an algorithm may be used in the construction of an $O(|V|\log|V|)$ algorithm for determining isomorphism of planar graphs [8]. The algorithm described here is not only theoretically optimal (to within a constant factor), but is practically useful. The algorithm has been implemented in Algol W and run on an IBM 360 model 65 computer. Experiments show that the algorithm can handle graphs with around 1000 edges in less than 10 seconds.

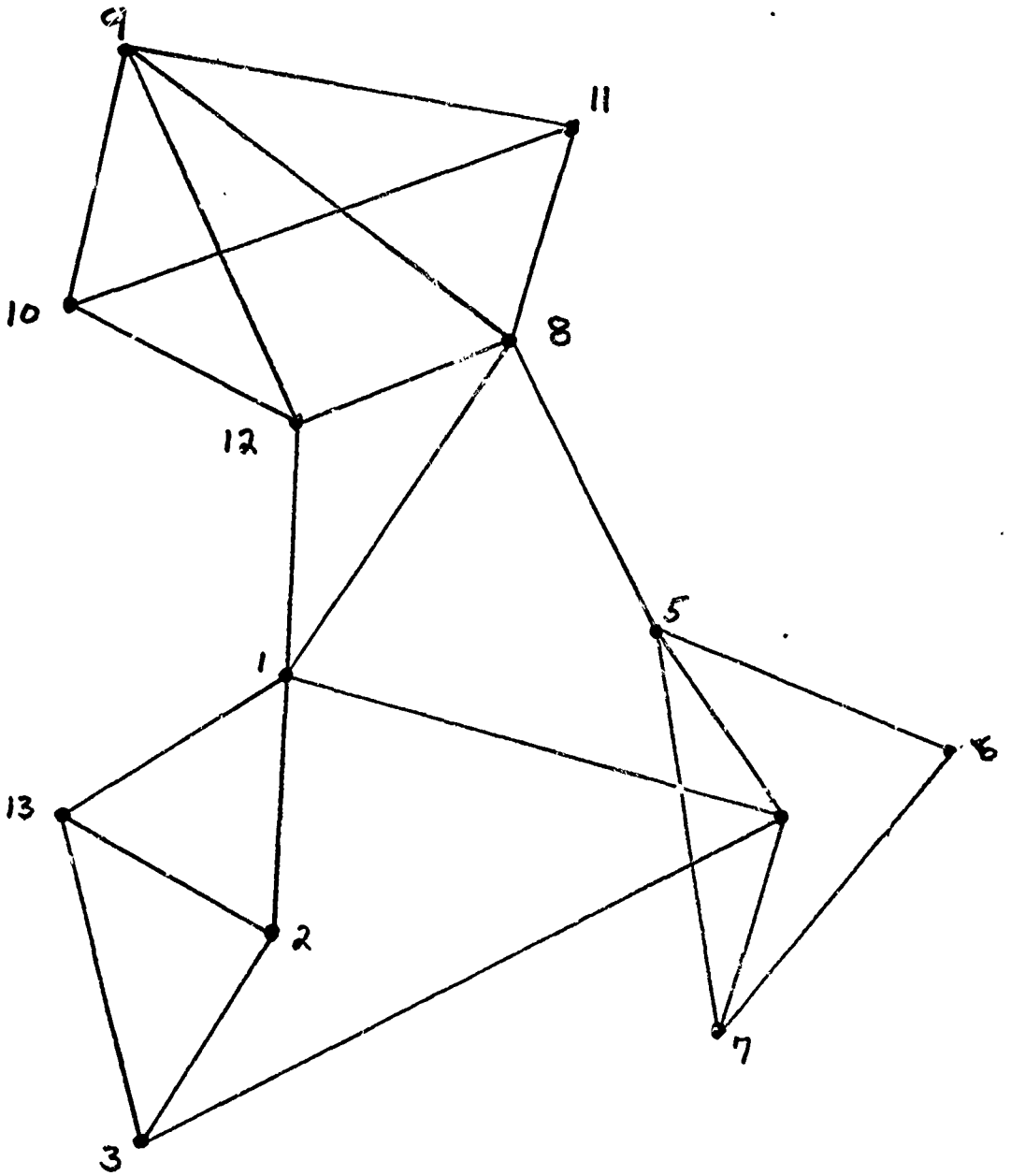



Fig 1: A biconnected graph G with separation pairs (1,4),(1,5),(4,5),(1,8),(4,8),(8,12).

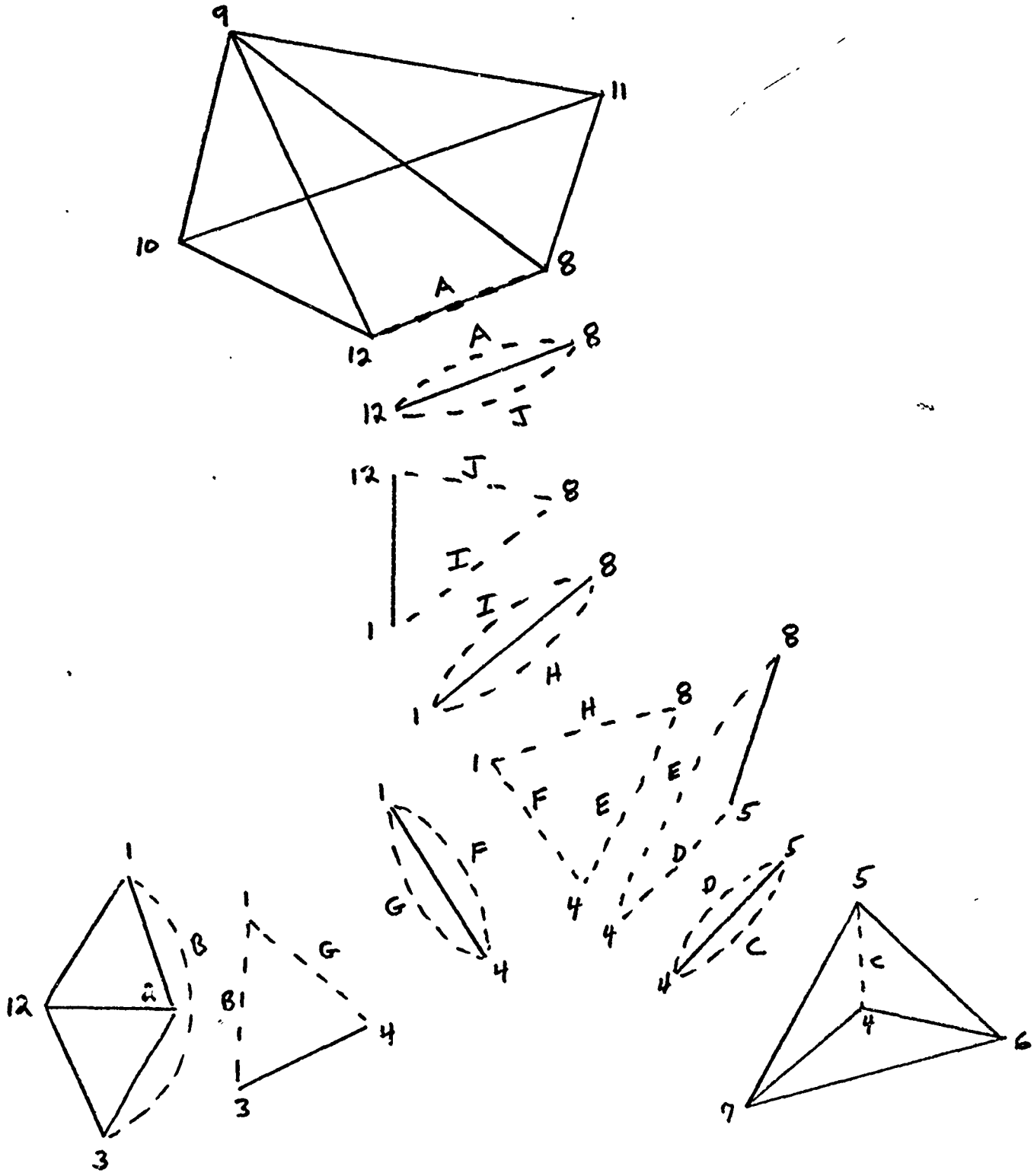



Fig. 2: The triconnected components of the graph G illustrated in Figure 1.

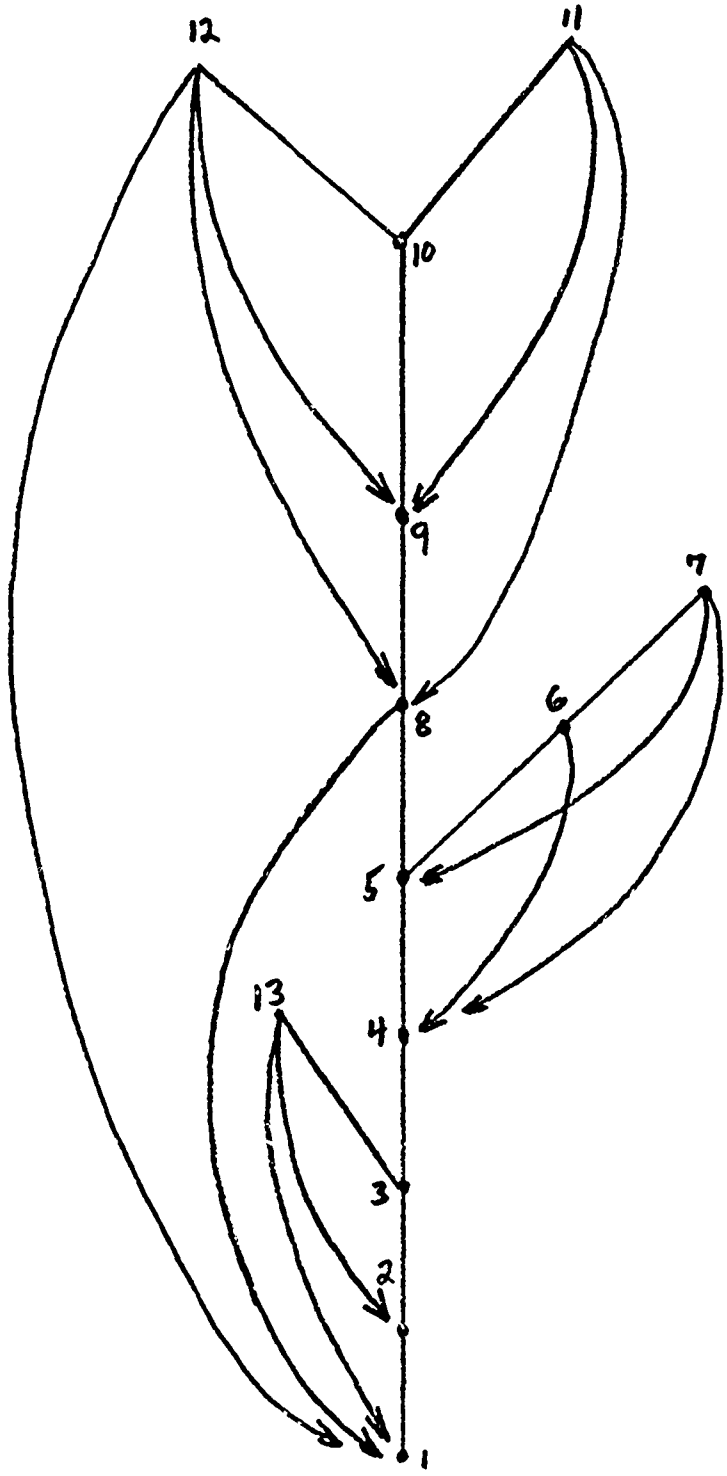



Fig. 3: Palm tree produced by a depth-first search of graph G illustrated in Figure 1.

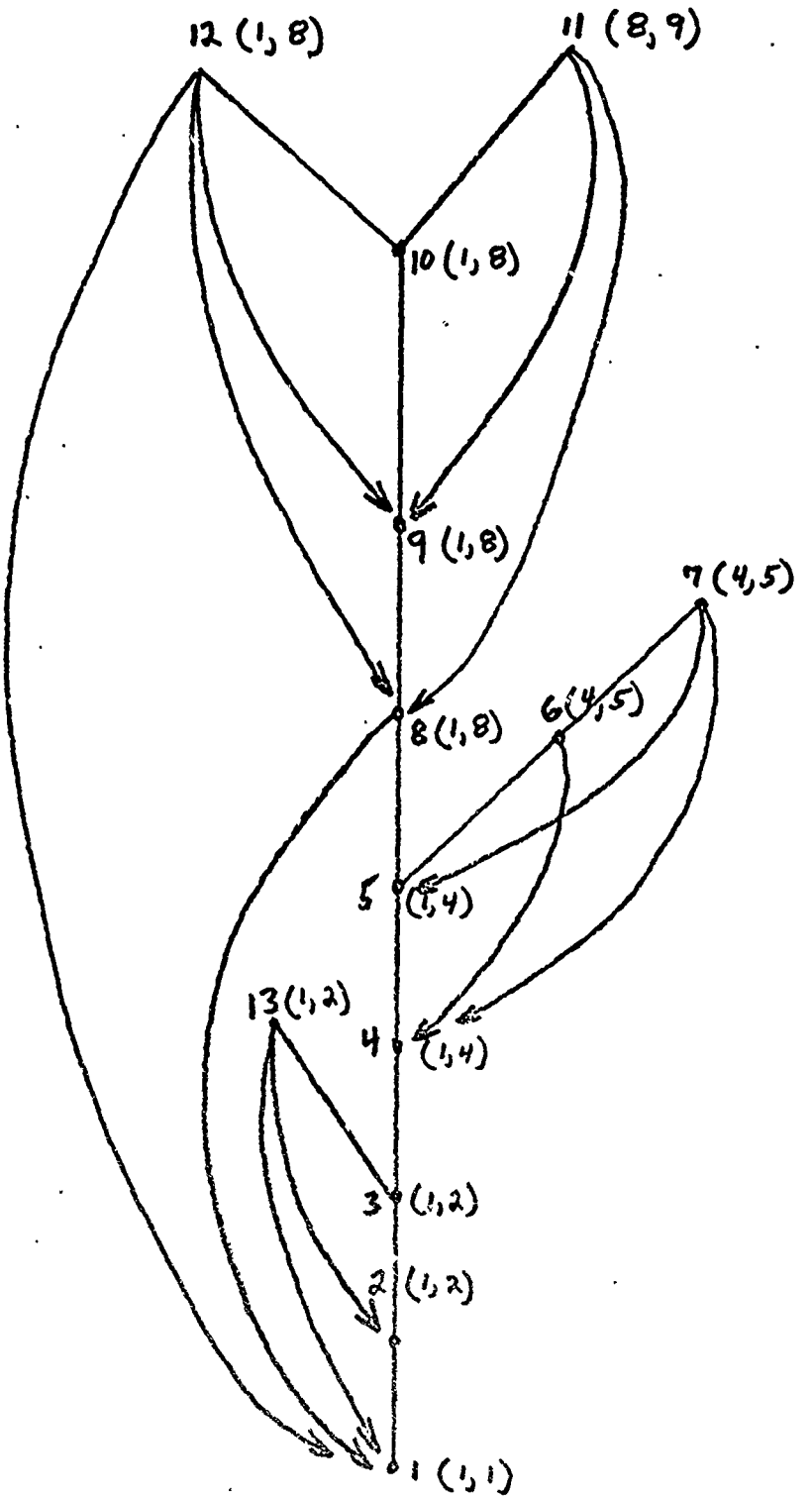



Fig 4: Ordered palm tree of graph G after second search with LOWPT1 and LOWPT2 values in parenthesis.

Type 1 pairs: (1,4),(1,5),(4,5),(1,8)
Type 2 pairs: (4,8),(8,12).

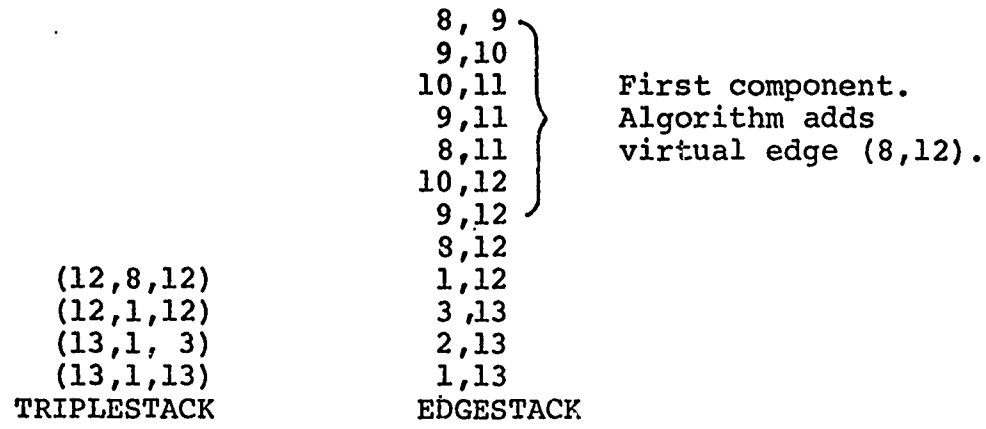



Fig 5: Contents of EDGESTACK and TRIPLESTACK when first biarticulation point pair (8,12) is detected.

## References


1. A. Ariyoshi, I. Shirakawa, and O. Hiroshi, "Decomposition of a Graph into Compactly Connected Two-Terminal Subgraphs", IEEE Trans on Circuit Theory Vol. CT-18, (1971) pp 430-435.

2. J. Bruno, K, Steiglitz and L. Weinburg, "A New Planarity Test Based on 3-Connectivity", IEEE Trans on Circuit Theory Vol. CT-17 (1970) pp 197-206.

3. G. Busacker and T. L. Saaty, Finite Graphs and Networks: An Introduction with Applications, McGraw-Hill, New York (1965).

4. S.A. Cook, "Linear Time Simulation of Deterministic Two-Way Pushdown Automata", IFIP Congress 71: Foundations of Information Processing. Ljubljana, Yugoslavia (August 1971). Amsterdam: North Holland Publishing Company., pp 174-179.

5. J. Edmonds and W. Cunningham, private communications.

6. F. Harary, Graph Theory, Addison-Wesley, Reading, Mass.(1969).

7. J.E. Hopcroft and R.E. Tarjan, "Efficient Algorithms for Graph Manipulation", CACM to appear.

8. J.E. Hopcroft and R.E. Tarjan, "Isomorphism of Planar Graphs (Working Paper)", Proceedings IBM Symposium on Complexity, to appear.

9. D.J. Kleitman, "Methods for Investigating Connectivity of Large Graphs" IEEE Trans on Circuit Theory Vol. CT-16 (1969) pp 232-233.

10. R.E. Tarjan, "Depth-First Search and Linear Graph Algorithms", SIAM Journal on Computing, to appear.

11. W.T. Tutte, Connectivity in Graphs, University of Toronto Press, 1966.